A Concurrency-Optimal List-Based Set

Vincent Gramoli[3] Petr Kuznetsov[1] Srivatsan Ravi[2] Di Shang[3]
[1]Télécom ParisTech
[2]TU Berlin
[3]NICTA and University of Sydney

February 6, 2015

Abstract

Designing a highly concurrent data structure is an important challenge that is not easy to meet. As we show in this paper, even for a data structure as simple as a linked list used to implement the *set* type, the most efficient algorithms known so far may reject correct concurrent schedules.

We propose a new algorithm based on a versioned try-lock that we show to achieve *optimal* concurrency: it only rejects concurrent schedules that violate correctness of the implemented type. We show empirically that reaching optimality does not induce a significant overhead. In fact, our implementation of the optimal algorithm outperforms both the Lazy Linked List and the Harris-Michael state-of-the-art algorithms.

1 Introduction

Multicore applications require highly concurrent data structures. Yet, the very notion of concurrency is vaguely defined, to say the least. What is meant by a "highly concurrent" data structure implementing a given high-level object type? Generally speaking, one could compare the concurrency of algorithms by running a game where an adversary decides on the schedules of shared memory accesses from different processes. At the end of the game, the more schedules the algorithm would accept without hampering high-level correctness, the more concurrent it would be. The algorithm that accepts all correct schedules would then be considered *concurrency-optimal*.

To illustrate the difficulty of optimizing concurrency, let us consider one of the most concurrency-friendly data structures [18]: the sorted linked list used to implement the integer set type. Since any modification on a linked list affects only a small number of contiguous list nodes, most of update operations on the list could, in principle, run concurrently without conflicts. For example, one of the most efficient concurrent list-based set to date, the Lazy Linked List [9], achieves high concurrency by holding locks on only two consecutive nodes when updating, thus accepting modifications of non contiguous nodes to be scheduled in any order. The Lazy Linked List is known to outperform the Java variant [12] of the CAS-based Harris-Michael algorithm [8, 15] under low contention because all its traversals, be they for read-only operations or to find the nodes to be updated, are *wait-free*, *i.e.*, they ignore locks and logical deletion marks. As we show below, the Lazy Linked List implementation is however not concurrency-optimal, raising two questions: Does there exist a more concurrent list-based set algorithm? And if so, does higher concurrency induce an overhead that precludes higher performance?

The concurrency limitation of the Lazy Linked List is caused by the locking strategy of its update operations: both insert(v) and remove(v) traverse the structure until they find a node whose value is larger or equal to v, at which point they acquire locks on two consecutive nodes. Only then the existence of the value v is checked: if v is found (resp. not found), then the insertion (resp., removal) releases the locks and returns without modifying the structure. By modifying metadata during lock acquisition without necessarily modifying the structure itself, the Lazy Linked List over conservatively rejects certain correct schedules.

To illustrate that the concurrency limitation of the Lazy Linked List may lead to poor scalability, consider Figure 1 that depicts the performance of a 100-element Lazy Linked List under a workload of 10% updates (insertions/removals) and 90% of contains on a 64-core machine. The list is comparatively small, hence all updates (even the failed insertions and removals) are likely to contend. We can see that when we increase the number of threads beyond 40, the performance drops significantly. This observation unveils an interesting desirable data structure property by which concurrent operations conflict on metadata only when they conflict

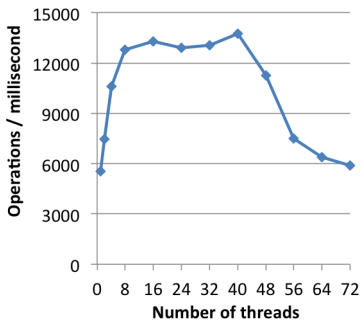

Figure 1: The concurrency limitation of the Lazy Linked List based set leads to poor scalability with only 10% updates as operations potentially contend on meta-data even when they do not modify the structure

on data. Note that this property extends the original notions of DAP [1, 7, 3] that are trivially ensured by most linked-list implementations simply because all their operations "access" the *head* node and, thus, are allowed to conflict on the meta-data.

Our main contribution is the *Versioned List*, the most concurrent (optimally concurrent, actually) and the most efficient list-based set algorithm to date. It exploits the logical deletion technique of Harris-Michael that divides the removal of a node into a logical and a physical step, and the wait-free traversal of the Lazy Linked List. In contrast to these techniques, it relies on a novel synchronization step inspired by transactional memory (TM): an update operation uses a CAS to set a versioned try-lock, based on the recent StampedLock of Java 8, immediately after the validation of the node succeeds[1]. If acquiring the try-lock fails, then the operation restarts.

We show that the resulting algorithm rejects a concurrent schedule only if otherwise the high-level correctness of the implemented set type (linearizability [13]) is violated. Our algorithm is thus provably concurrency-optimal: no other correct list-based set algorithm can accept more schedules.

The evaluation of our versioned list shows that achieving optimal concurrency does not necessitate a costly overhead. Extensive experiments on two 64-way multi-core architectures (x86-64 and SPARC) confirmed that the Versioned List outperforms the state-of-the-art algorithms [9, 12]. In particular, as our algorithm differs from the Lazy Linked List by validating before locking, it outperforms the Lazy Linked List performance by 3.5× for 64 threads on the workload of Figure 1. In addition, as our algorithm differs from Harris-Michael by avoiding metadata accesses during traversals, it outperforms the Java variant of Harris-Michael's (even with its RTTI optimization [9]) by up to 2.2× on read-only workloads.

In the rest of the paper, we describe our system model (Section 2), present the Versioned List and prove it correct (Section 3). We show it concurrency-optimal as opposed to previous work (Section 4). We evaluate its performance in Section 5. Finally, we discuss the related work (Section 6) and conclude (Section 7). The sequential specification of the set type and the missing proofs are deferred to Appendices A and B, respectively.

2 Preliminaries

Objects and implementations. We consider a standard asynchronous shared-memory system, in which $n > 1$ processes $p_1, \ldots, p_n$ communicate by applying operations on shared *objects*. An object is an instance of an *abstract data type* that specifies the set of operations the object exports, the set of responses the operations return, the set of states the object can take, and the sequential specification that stipulates the object's correct sequential behavior. To *implement* a *high-level* object from a set of shared *base* objects, processes follow an *algorithm*, which is a collection of deterministic state machines, one for each process. The algorithm assigns initial values to the base objects and processes and specifies the base-object operations a process must perform when it executes every given operation. To avoid confusion, we call operations on the base objects *primitives*. A primitive is an atomic *read-modify-write (rmw)* on a base object [11] characterized by a pair of deterministic

[1]The possibility of "pre-locking validation" was suggested in [9], but to the best of our knowledge, no algorithm was proposed to implement it.

functions $\langle g, h\rangle$: given the current state of the base object, g is an *update function* that computes its state after the primitive is applied, while h is a *response function* that specifies the outcome of the primitive returned to the process. Special cases of rmw primitives are *read* (g leaves the state unchanged and h returns the state) and *write* (g updates the state with its argument and h returns *ok*).

Executions. An *event* of a process p_i is an invocation or response of an operation performed by p_i on a high-level object implementation, a rmw primitive $\langle g, h\rangle$ applied by p_i to a base object b along with its response r (we call it a *rmw event* and write $(b, \langle g, h\rangle, r, i)$). A *configuration* specifies the value of each base object and the state of each process. The *initial configuration* is the configuration in which all base objects have their initial values and all processes are in their initial states.

An *execution fragment* is a (finite or infinite) sequence of events. An *execution* of an implementation I is an execution fragment where, starting from the initial configuration, each event is issued according to I and each response of a rmw event $(b, \langle g, h\rangle, r, i)$ matches the state of b resulting from all preceding events. We assume that executions are *well-formed*: no process invokes a new high-level operation before the previous high-level operation returns.

Let $\alpha|p_i$ denote the subsequence of an execution α restricted to the events of process p_i. Executions α and α' are *equivalent* if for every process p_i, $\alpha|p_i = \alpha'|p_i$. An operation π *precedes* another operation π' in an execution α, denoted $\pi \rightarrow_\alpha \pi'$, if the response of π occurs before the invocation of π' in α. Two operations are *concurrent* if neither precedes the other. An execution is *sequential* if it has no concurrent operations. An operation is *complete* in α if the invocation event is followed by a *matching* response; otherwise, it is *incomplete* in α. Execution α is *complete* if every operation is complete in α.

High-level histories and linearizability. A *high-level history* $\tilde{H}$ of an execution α is the subsequence of α consisting of all invocations and responses of (high-level) operations.

A complete high-level history $\tilde{H}$ is *linearizable* with respect to an object type τ if there exists a sequential high-level history S equivalent to $\tilde{H}$ such that (1) $\rightarrow_{\tilde{H}} \subseteq \rightarrow_S$ and (2) S *is consistent with the sequential specification of type* τ.

Now a high-level history $\tilde{H}$ is linearizable if it can be *completed* (by adding matching responses to a subset of incomplete operations in $\tilde{H}$ and removing the rest) to a linearizable high-level history [13, 2].

Sequential implementations. A *sequential implementation* of an object type τ specifies, for each operation of τ, a deterministic procedure that performs *read* and *write* primitives on a collection of base objects that encode the state of the object, and returns a response, so that the specification of τ is respected in all sequential executions.

Consider the conventional set type exporting insert, remove, and contains operations with standard sequential semantics: insert(v) adds v to the set and returns true if v is not already there, and returns false otherwise; remove(v) drops v from the set and returns true if v is there, and returns false otherwise; and contains(v) returns true if and only if v is in the set. The exact specification and the list-based sequential implementation of set are presented in the appendix (Algorithm 3).

3 The Versioned List Set

In this section, we describe our *Versioned List* implementation of the set (Algorithm 1) and prove it linearizable.

List nodes. Each *node* in the list has 4 fields as depicted in Lines 2–13: *val* stores the value of the node, *next* is a pointer/reference to the next node, *deleted* is a boolean marker initialized to false, which indicates whether a node has been removed from the list, and the *versioned try-lock* is described below.

The versioned try-lock. The versioned try-lock is defined as a pair $\langle ver, lock\rangle$ of a version number and a boolean locking state, that can be modified together atomically. The versioned try-lock (Algorithm 2) supports the following operations:

- getVersion(): returns the current version.
- tryLockAtVersion(ver): tries to change the locking state from false to true atomically if the current version number matches ver. The method fails and returns false if either the node is already locked or the version

```
1:  Shared variables:
2:    node is a record with fields:
3:      val, its value
4:      next, its reference to the next node in the list
5:      deleted, a boolean indicating whether the node is
6:        logically deleted
7:      vlock, a versioned lock: counter whose least significant
8:        bit is a lock
9:    Initially the list contains only two nodes head, tail,
10:     head.val = −∞, tail.val = +∞
11:     head.next = tail
12:     head.deleted = tail.deleted = false
13:     head.vlock = tail.vlock = ⟨0, false⟩

14: contains(v):                                   ▷ wait-free contains
15:   curr ← head
16:   while curr.val < v do
17:     curr ← curr.next
18:   return (curr.val = v ∧ ¬curr.deleted)

19: validate(v, prev):
20:   pVer ← prev.vlock.getVersion()          ▷ return lock version
21:   if prev.deleted then return ⊥                    ▷ full abort
22:   curr ← prev.next
23:   while curr.val < v do
24:     pVer ← curr.vlock.getVersion()
25:     if curr.deleted then goto line 20              ▷ partial abort
26:     prev ← curr          ▷ the above line checks prev.deleted
27:     curr ← curr.next              ▷ same as reading prev.next
28:   return ⟨prev, pVer, curr⟩

29: waitfreeTraversal(v):      ▷ wait-free traversal used in updates
30:   prev ← curr ← head
31:   while (curr.val) < v do                ▷ until position is reached
32:     prev ← curr              ▷ keep track of the previous node
33:     curr ← curr.next
34:   return prev

35: insert(v):
36:   prev ← waitfreeTraversal(v)
37:   if (⟨prev, pVer, curr⟩ ← validate(v, prev)) = ⊥ then
38:     goto line 36          ▷ full abort: restart from beginning
39:   if curr.deleted then goto line 37
40:   if curr.val = v then return false   ▷ v already in the set
41:   newNode.val ← v          ▷ allocate a new node with value v
42:   newNode.next ← curr
43:   if ¬prev.vlock.tryLockAtVersion(pVer) then        ▷ v.-lock
44:     goto line 37                                  ▷ partially abort
45:   prev.next ← newNode
46:   prev.vlock.unlockAndIncrementVersion()
47:   return true

48: remove(v):
49:   prev ← waitfreeTraversal(v)
50:   if (⟨prev, pVer, curr⟩ ← validate(v, prev)) = ⊥ then
51:     goto line 49          ▷ full abort: restart from beginning
52:   if (curr.val ≠ v ∨ curr.deleted) then return false
53:   if ¬prev.vlock.tryLockAtVersion(pVer) then        ▷ v.-lock
54:     goto line 50                                  ▷ partially abort
55:   curr.vlock.lockAtCurrentVersion()                  ▷ spin lock
56:   curr.deleted ← true                             ▷ logical delete
57:   prev.next ← curr.next                          ▷ physical delete
58:   curr.vlock.unlockAndIncrementVersion()
59:   prev.vlock.unlockAndIncrementVersion()
60:   return true
```

Algorithm 1: The versioned list-based set

does not match; otherwise it succeeds and returns true.

– lockAtCurrentVersion(): spins until it acquires the lock with the current version.

– unlockAndIncrementVersion(): only called by the process that previously acquired the lock via successful tryLockAtVersion(ver) or lockAtCurrentVersion(). It unconditionally sets the locking state from true to false and increments the current version number atomically from $\langle ver, \text{true}\rangle$ to $\langle ver{+}1, \text{false}\rangle$.

In our Java 8 implementation of the versioned try-lock (Algorithm 2), we tested a single integer variable AtomicInteger that supports single-word CAS as well as the more recent StampedLock[2]. One could alternately use the least significant bit of an integer on x86 architectures to represent the locking state where 0 means "unlocked" and 1 means "locked", hence losing portability. Distinct version numbers are represented by all the even values: to extract the version we use a bit-mask that always sets the last bit to 0 when doing a bitwise *and*.

We now describe our list-based set implementation. Recall that the *set* type exports operations insert(v), remove(v) and contains(v), with $v \in \mathbb{Z}$ (see the appendix for a detailed specification). The list is initialized with 2 nodes: *head* (storing the minimum sentinel value) and *tail* (storing the maximum value), *head.next* storing the pointer to *tail*.

Contains. The algorithm for contains simply traverses in the *wait-free* manner, exactly as in the sequential algorithm (Algorithm 3 of Appendix A) except that in the end, we also check that *curr* is not deleted (Line 18).

[2] https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/StampedLock.html

```
1: Private field:
2:   lockValue, an integer that supports CAS
3:   Initially lockValue = 0

4: getVersion():                                  ▷ only return even value
5:   return lockValue.read() & 1111...1110        ▷ bitwise

6: tryLockAtVersion(ver):                         ▷ assuming ver is even
7:   return lockValue.CAS(ver, ver + 1)           ▷ next odd value

8: lockAtCurrentVersion():                        ▷ spin lock on the latest version
9:   success ← false
10:  while (¬success) do                          ▷ spin until we get the lock
11:    ver ← getVersion()
12:    success ← lockValue.tryLockAtVersion(ver)

13: unlockAndIncrementVersion():                  ▷ assuming locked (odd val)
14:  val ← lockValue.read()
15:  lockValue.CAS(val, val + 1)                  ▷ use atomicSet if available
```

Algorithm 2: The CAS-based implementation for the versioned try-lock

This wait-free traversal, introduced by Heller *et al.* [9], results in a highly efficient contains algorithm, as its only overhead (compared to the sequential implementation) comes from a single memory read, on *curr.deleted*.

Pre-locking validation in insert and remove. For insert and remove, we first traverse the list in a wait-free manner (Lines 29–34) until we find the position where a node might be inserted or deleted, i.e., where $prev.val < v$ and $curr.val \geq v$. Then we use the novel technique of *pre-locking-validation*: it *validates* the state of the nodes prior to locking. Generally, an optimistic lock-based algorithm follows this pattern for updates:

```
1: read data on node
2: lock node
3:     re-read & validate integrity: if fail then unlock & restart
4:     modify data
5: unlock node
```

With pre-locking-validation, the new pattern becomes:

```
1: read node version ver
2: read & validate data integrity: if fail then restart
3: try-lock-at-version(ver): if fail then restart
4:     modify data
5: unlock-and-increment-version
```

The reason why we can validate *before* acquiring the lock is that the consistency of the validation result is protected by the version number. Observe that in this new pattern, any modification to the node first acquires a lock and the version is changed only when releasing the lock. Thus, in case any concurrent thread is modifying or has modified the node between the read-version and the try-lock-at-version steps, the try-lock will fail either because of a lock conflict or a new version number.

The validate function. The validate function (Lines 20-28) invoked by our update operations is a short traversal that stores the version of *prev*, then checks that *prev* is not logically deleted and finally sets *curr* to *prev.next*. The validation conditions are (1) *prev* is not deleted ($prev.deleted = \text{false}$), and (2) *prev.next* points to *curr* ($prev.next = curr$).

Note that after the traversal completes and before the validation starts, some new node could be inserted between *prev* and *curr* or *curr* could be deleted. Instead of using the *curr* node from the traversal to check whether $prev.next = curr$ we simply re-traverse from the *prev* node.

During validation and locking, we might fail due to conflicts with a concurrent operation, in which case we need to abort and restart the operation. We included an optimization of *partial abort* where instead of restarting from *head*, we only need to restart from *prev* under the condition that *prev.deleted* is not true (we already know that $prev.val < v$). As a result of this "versioned-traversal", we get *prev* and *curr* together with *prev*'s version $pVer$ (Line 28): if the operation later successfully locks *prev* at $pVer$, we are sure that *prev* is not deleted and $prev.next = curr$. Finally, after the validation, we check *curr.val* to see if the value we are trying to insert or remove is present in (or absent from) the set.

Inserting a node. For insert, we create the new node *before* entering the critical section (Line 41): the reason here is that we want to optimize concurrency and minimize the length of the critical section. However, it is possible for our implementation to execute the node creation (Line 41) multiple times since we could potentially

abort later and restart. A possible optimization is to keep track of the *newNode* reference and make a check before Line 41 so that we allocate the memory at most once per insert operation. We enter the critical section by trying to lock *prev* at version $pVer$ atomically in Line 43 ($pVer$ is the version of the *prev* node obtained after validation in Line 37). If we obtain the lock successfully, it implies that we are already in a valid state (conditions (1) and (2) are satisfied). Once we have successfully acquired the lock on *prev*, we link-in the new node (Line 45).

Removing a node. For remove, we also require the lock on *curr*, which is obtained using the spin-lock in Line 55). We need to lock *prev* at version $pVer$ for the same reason as insert: to make sure no concurrent thread is inserting a node between *prev* and *curr* or deleting the *curr* node. Additionally, we need to lock *curr* at its current version to prevent concurrent threads from inserting/deleting the node after *curr*. Removing a node now involves two steps: a logical delete that sets *curr.deleted* to true (Line 56) and a physical delete that changes *prev.next* (Line 57).

Finally, we exit the critical section by releasing the locks on the node(s) involved, increment the version in one atomic step (Line 46, 58, 59) and return true for the operation.

Progress. It is easy to see that the contains operation is *wait-free*: a matching response is returned within a finite number of its events. The update operations ensure *deadlock-freedom*: assuming no process fails in the critical section, some process makes progress by completing each of its operations.

Proof of linearizability. We now show that the Versioned List algorithm is linearizable with respect to the *set* type. Let α be a finite execution of Algorithm 1 and $<_\alpha$ denote the total-order on events in α. For the sake of the proof, we assume that α starts with an artificial sequential execution of an insert operation π_0 that inserts *tail* and sets $head.next = tail$. Let H be the high-level history exported by α.

Completions. We obtain a completion $\tilde{H}$ of H as follows. The invocation of an incomplete contains operation is discarded. The invocation of an incomplete $\pi =$ remove operation that has not performed the write in Line 56 is discarded; otherwise, it is completed with response true. The invocation of an incomplete $\pi =$ insert operation that has not performed the write in Line 45 is discarded; otherwise, it is completed with response true.

Linearization points. We obtain a sequential high-level history $\tilde{S}$ equivalent to $\tilde{H}$ by associating a linearization point ℓ_π with each operation π as follows.

For every $\pi =$ insert(v) that returns true in $\tilde{H}$, ℓ_π is associated with the write event in Line 45 (rendering the node that stores v reachable from the *head*); otherwise ℓ_π is associated with the last read of a node's *next* field performed by π in α.

For every $\pi =$ remove(v) that returns true in $\tilde{H}$, ℓ_π is associated with the write event in Line 56 (setting the *deleted* flag of a list's element); otherwise ℓ_π is associated with the last read of a node's *next* field performed by π in α.

For $\pi =$ contains(v) that returns true, ℓ_π is associated with the last read performed by π in which π finds the *deleted* field of a reachable node storing v be false (Line 18).

For $\pi =$ contains(v) that returns false in H,

- if π reads ($X.value \neq v$) in Line 18, where X is the last node read by π in α, ℓ_π is assigned to the read of the *next* field of the node accessed by π immediately before X
- if π reads ($X.value = v$) in Line 18, where X is the last node read by π in α, ℓ_π is defined as follows: let π_1 be the remove operation that performs the last write to $X.deleted$ (Line 56) prior to the read of $X.deleted$ by π. Then, ℓ_π is chosen to be the first event performed by π_1 immediately after the write to $X.deleted$, but prior the read of $X.deleted$ by π. Otherwise if no such event of π_1 exists, then ℓ_π is the read of $X.deleted$ by π.

Since linearization points are chosen within the intervals of operations performed in α, for any two operations π_i and π_j in $\tilde{H}$, if $\pi_i \rightarrow_{\tilde{H}} \pi_j$, then $\pi_i \rightarrow_{\tilde{S}} \pi_j$. Intuitively, the linearization point of each insert (resp., remove) operation determines the instance when the operation takes effect, *i.e.*, the corresponding element becomes reachable (resp., unreachable). A successful contains(v) operation is linearized at the moment an "undeleted" list element storing v has been reached. A failed contains operation is linearized at the moment it detects that no "undeleted" node storing v can be reached.

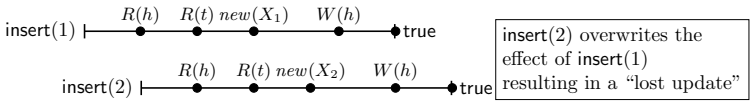

Figure 2: A history exporting an observably incorrect schedule σ; for succinctness, $R(h)$ and R(t) refers to reads of both *val* and *next* fields; $W(h)$ refers to write on *head.next*

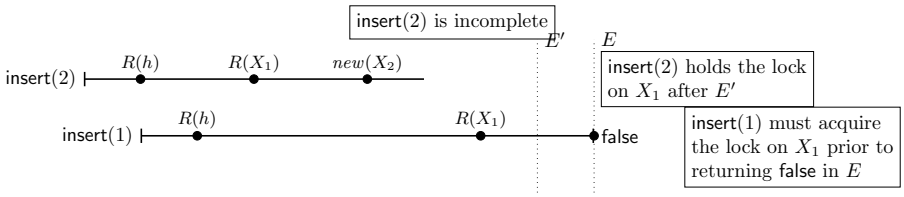

Figure 3: A schedule rejected by the lazy linked list; initial state of the list is $\{X_1\}$ that stores value 1; $R(X_1)$ refers to reads of both the *val* and *next* fields; $new(X_2)$ creates a new node storing value 2

Thus, we can prove the linearizability of the versioned list w.r.t the set type (the proof is given in Appendix B):

Theorem 1. *Versioned List is linearizable with respect to the set type.*

4 Concurrency analysis

To characterize the ability of a concurrent implementation to process arbitrary interleavings of sequential code, we introduce the notion of a *schedule*. Intuitively, a schedule of an execution of a list-based set algorithm specifies the order in which high-level operations access the nodes of the list. List-based set algorithms generally follow the sequential implementation (denoted *LL*) of operations *insert*, *remove* and *contains*: every high-level operation *reads* the nodes sequentially until the desired fragment of it is located. The update operation (*insert* or *remove*) then *writes* to the *next* field of one of the nodes the address of a new node (if it is *insert*) or the address of the node that follows the removed node in the list (if it is *remove*). (The sequential write can be implemented using a *CAS* primitive [8].) For the detailed pseudocode of the sequential implementation followed by the concurrent linked-list implementations, we refer to Algorithm 3 in the appendix.

4.1 Schedules and local serializability

An execution of our concurrent implementation involves reading and writing to the nodes fields *val* and *next*, as well as reading and modifying *meta-fields* such as *deleted*, and *vlock* (cf. Section 3). Naturally, we identify the events in the execution of the concurrent implementation corresponding to the "sequential" *reads*, *writes* (of *val* and *next* fields) and *node creation* events (Line 12 in the sequential implementation *LL*) as marked explicitly.

Let α be an execution of our concurrent implementation. We define the *history of an execution* α as the subsequence of α corresponding to the events that "take effect". Formally, for every update operation π in α, $H|\pi$ is defined to be the subsequence $\alpha|\pi$ consisting of the reads, writes and node creation events from the last invocation of the function waitfree-traversal by π in Lines 36 and 49. For every contains operation π in α, $H|\pi$ is defined to be the subsequence $\alpha|\pi$ consisting of the reads and writes on a node's *val* and *next* fields.

Intuitively, a *schedule* corresponds to some interleaving of the sequential reads, writes, node creation events and invocation and responses of high-level operations performed in the sequential implementation *LL*. Formally, a *schedule* is an equivalence class of histories that agree on the order of reads, writes, node creation events and high-level operations, but not neccesarily on the responses of high-level operations and read events. Observe that, in our concurrent implementation, every read operation (on a base object x) returns the argument of the latest preceding write (on x). Thus, for every history, there exists exactly one schedule.

Definition 1. *We say that a schedule σ is* locally serializable *(with respect to the sequential implementation of list-based set LL) if for each of its operations π, there exists a history S of LL such that $\sigma|\pi = S|\pi$.*

Definition 2. *We say that a schedule is* correct *if it is (1) linearizable (with respect to the* set *type), (2) locally serializable (with respect to LL).*

Theorem 2 (Correctness)**.** *The Versioned List implementation accepts only correct schedules.*

Proof. Take any schedule σ of Algorithm 1. Theorem 1 implies that the high-level history of σ is linearizable with respect to the *set* type.

To show local serializability, we first remark that every operation traverses the list starting from the *head* node and reads the *next* field of a node to locate the subsequent node. Before adding a new node to the list (Line 45), each insert operation initializes the node's *val* and *next* filed, so that at all times the *next* field of a node stores a pointer to an inserted node with a strictly higher value or to the *tail* node. Furthermore, the values stored in the list are integers, every operation invoked with parameter v eventually locates the node storing v or a higher value. Thus, every sequence of non-aborted events of every operation π is finite. Hence, there exists a sequence of insert operations S_0, such that $S_0 \cdot \sigma|\pi$ is a sequential history of *LL*. □

4.2 Optimal concurrency

We show that any finite schedule rejected by our algorithm is not *observably correct*. A correct schedule σ is observably correct if by completing update operations in σ and extending, for any $v \in \mathbb{Z}$, the resulting schedule with a complete sequential execution contains(v), applied to the resulting contents of the list, we obtain a correct schedule. Here the contents of the list after a given correct schedule is determined based on the order of its write operations. For each node, we define the resulting state of its *next* field based on the last write in the schedule. Since in a correct schedule each new node is first created and then linked to the list, we can reconstruct the *state of the list* by iteratively traversing it, starting from *head*.

Intuitively, a schedule is observably correct if it incurs no "lost updates". Consider, for example a schedule (cf. Figure 2) in which two operations, insert(1) and insert(2) applied to the initial empty set. Imagine that they first both read *head*, then both read *tail* and then both perform writes on the *head.next*. The resulting schedule is trivially correct (both operations return *true* so the schedule can some from a complete linearizable history). However, in the schedule, one of the operations, say insert(1), overwrites the effect of the other one. Thus, if we extend the schedule with a complete execution of contains(2), the only possible response it may give is *false* which obviously does not produce a linearizable high-level history.

Theorem 3 (Optimality)**.** *Versioned List accepts all observably correct schedules.*

Proof. Let σ be any schedule of our concurrent implementation. Recall that, for every update operation π in σ, $\sigma|\pi$ is defined to be the subsequence $\sigma|\pi$ consisting of the reads, writes and node creation events from the last invocation of the function waitfree-traversal by π in Lines 36 and 49; and for every $\pi =$ contains in σ, $\sigma|\pi$ is the subsequence $\sigma|\pi$ consisting of the reads and writes on a node's *val* and *next* fields.

We prove that any schedule *rejected* by our algorithm is not observably correct. More precisely, we show that an operation restarts a fragment of it execution (in Lines 44, 54) only if extending it with a read or a write on *next* or *val* fields would result in schedule that is not observably correct.

We first observe that if a node is logically deleted (Line 56), then its next write renders the node unreachable from the *head* node (Line 57). Thus, an update operation π partially restarts because of reading a logically deleted node (Lines 38, or 51) only if it is concurrent with a *remove* operation which, when completed would physically remove the node addressed by π at the end of its traversal. It is easy to see that regardless of what this operation π is (insert(v) or remove(v)), if we complete it in turn and then extend the resulting schedule with contains(v), the effect of π will not be seen and the schedule will not be linearizable.

Similarly, an update operation π partially restarts in Lines 44, 54 after finishing its traversal phase, if it fails in grabbing a lock on one of the nodes it is about to modify. Thus, π is concurrent with another update operating on the same node. Again, by completing both π and the concurrent update, we obtain a schedule in which one of the updates is "lost", so that its extension with some contains(v) will not be linearizable. □

Theorem 3 shows that our implementation only rejects concurrent schedules that would result in a violation of linearizability. On the other hand, we can easily describe observably correct schedules that are rejected by the Lazy Linked list [9] and Harris-Michael Linked list implementations [8, 15].

The Lazy Linked List. Our example illustrates how the post-locking validation strategy employed by the Lazy Linked list makes it sub-optimal w.r.t concurrency. As explained in the introduction, the insert operation of the

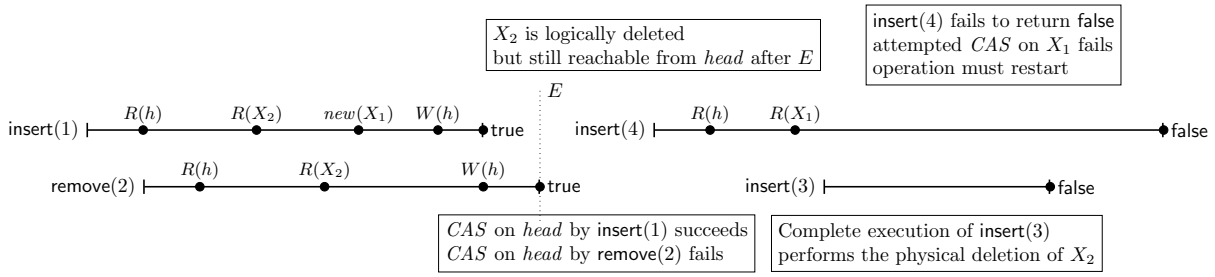

Figure 4: A schedule rejected by the Harris-Michael linked list; initial state of the list is $\{X_2, X_3, X_4\}$; each X_i stores value i; $R(X)$ refers to reads of both the *val* and *next* fields; $W(h)$ is *CAS* that attempts to set *head.next* to the desired node if it has not changed since the previous read

Operation segment	Algorithm	Number of concurrent threads											
		1	2	4	8	16	24	32	40	48	56	64	72
Traversal	Lazy linked list	2.9	8.9	16.4	30.2	58.5	65.5	61.0	61.0	71.7	100.4	75.4	68.9
	Harris-Michael	5.2	9.9	15.9	29.2	59.0	91.4	127.9	153.3	177.1	203.2	228.9	252.8
	Versioned list	3.7	7.2	12.3	22.8	42.3	61.5	79.5	89.2	96.0	109.6	122.2	202.9
Update	Lazy linked list	3.7	11.6	17.1	26.7	88.8	211.1	503.0	1019.5	1355.7	1814.6	2163.9	2634.2
	Harris-Michael	1.0	3.8	4.8	5.4	6.4	7.0	8.3	9.4	10.2	11.1	12.9	13.7
	Versioned list	2.3	3.8	4.5	5.4	6.4	7.3	8.0	9.2	10.9	12.9	16.2	162.6

Table 1: The relative time spent on list traversal and node update per operation on average using the benchmark with size 100 and update ratio 100%

Lazy Linked list acquires the lock on the nodes it writes to, prior to the check of the node's state. Consider the schedule depicted in Figure 3: insert(2) traverses the list, reaches node X_1 storing value 1, acquires the lock on X_1 and creates a new node that stores value 2. Observe that, at this point in the execution, the implementation has not performed the write to X_1 (corresponding to the write in the sequential implementation *LL* in Line 13) and thus, must hold the lock on X_1 after E'. However, insert(1) must also acquire the lock on X_1 prior to returning a matching response false. But it cannot do so until insert(2) releases the lock on X_1. Consequently, the Lazy Linked list cannot export the schedule depicted in Figure 3.

The Harris-Michael linked list. In the Harris-Michael linked list (cf. [12, Chapter 9]), each update operation attempts to physically remove (using *CAS*) nodes that are marked for deletion as it traverses the list. If the attempt fails, the operation is restarted. Figure 4 depicts a schedule σ that is rejected by the Harris-Michael algorithm. The initial state of the list $\{X_2, X_3, X_4\}$, where each X_i stores value i. First insert(1) runs concurrently with a remove(2), where insert(1) performs a *CAS* on *head* to set *head.next* to X_1, after which the remove(2) performs a logical deletion of X_2 (by setting a *deleted* flag) and then invokes *CAS* to set *head.next* to X_3. However, this *CAS* fails, and the operation returns true after having only logically deleted X_2. Thus, at the end of this execution, X_2 is still reachable from the *head*. We now extend this execution with an insert(4) that reads *head*, X_1 and prior to the attempted physical removal of X_2, a concurrent insert(3) performs this physical removal, thus forcing insert(4) to restart. Therefore, Harris-Michael implementation cannot accept σ (clearly, accepted by Versioned List).

5 Experimental evaluation

We compared our versioned list in Java to the lock-based Lazy Linked List [9] and Harris-Michael's non-blocking list [8, 15] with its wait-free and RTTI optimization suggested in Java by Heller et al. [9] using the Synchrobench benchmark suite [4]. For the versioned list, we tested both a hand-crafted versioned lock and one implemented on top of the Java 8 StampedLock and only report the better results of the latter. The source code is publicly available as part of Synchrobench at `https://github.com/gramoli/synchrobench/tree/master/java/src/linkedlists`.

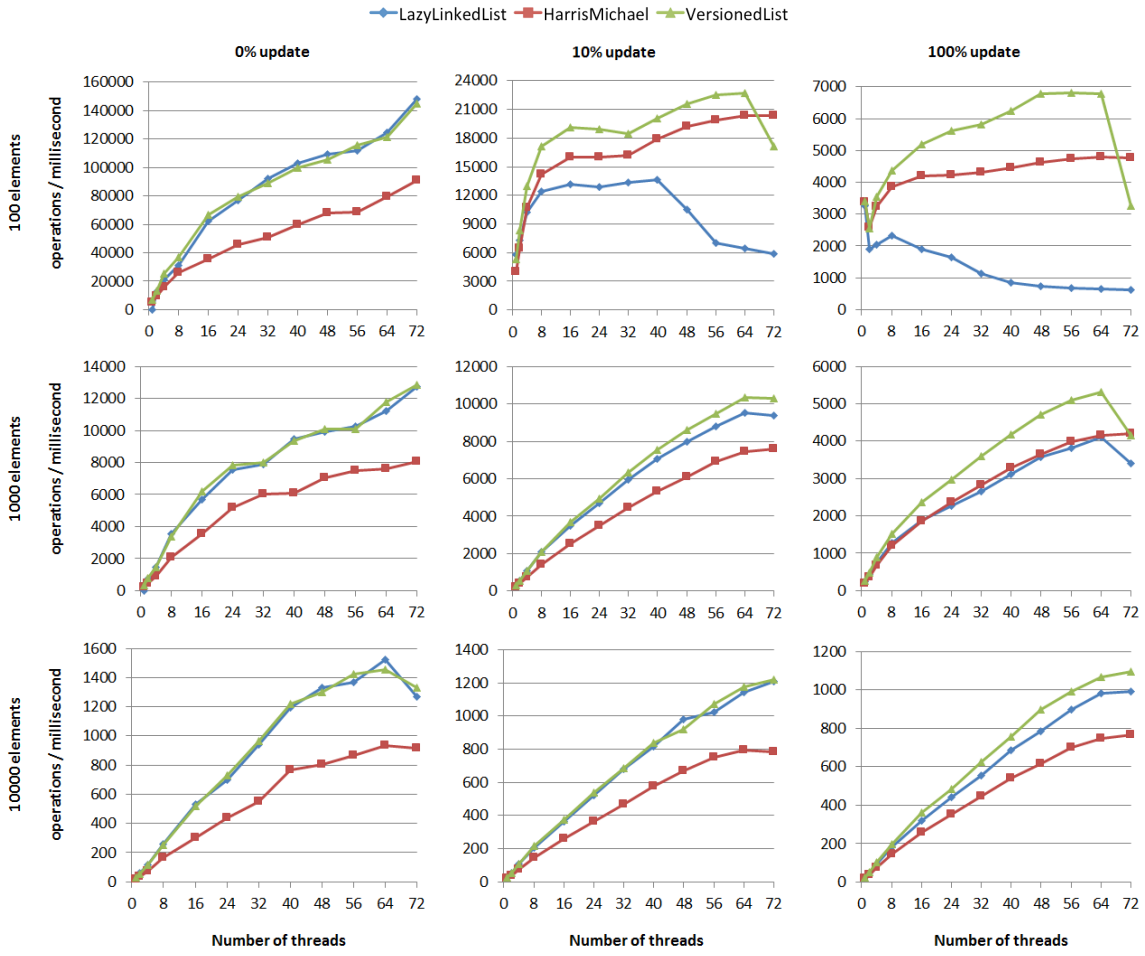

Figure 5: The performance results obtained on the x86 architecture

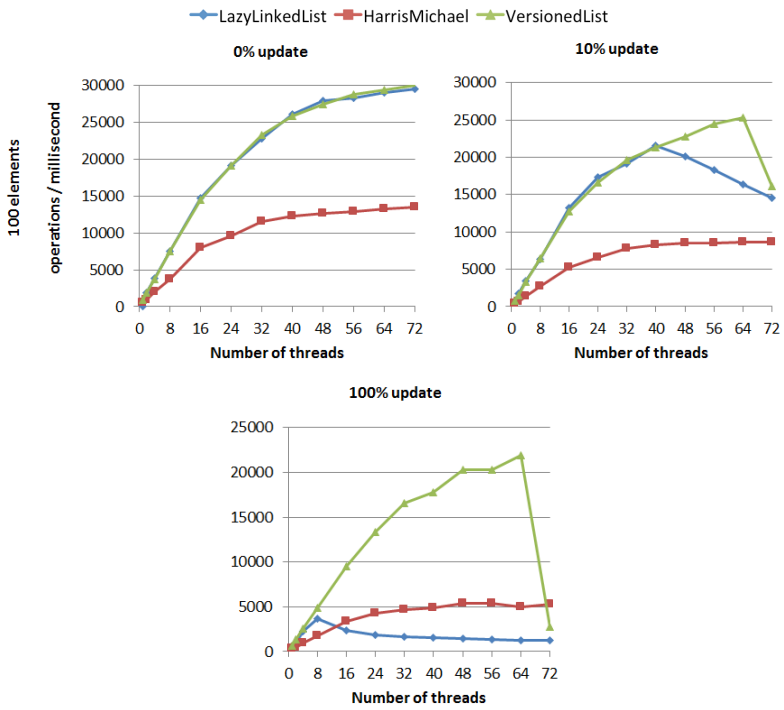

Figure 6: The performance results obtained on the SPARC architecture

5.1 Setup

We report the performance results of the three implementations obtained from two different architectures (x86 and SPARC). More precisely, we performed the experiment on a 4 socket AMD Opteron 6378 2.4 GHz 16-core (64-core in total) running Linux Fedora 18 and on a Sun Niagara 2 running SunOS 5.1 with 8 cores each running 8 simultaneous multiple threads at 1.165 GHz (64 hardware threads in total). Both architectures run the 64-bit Java HotSpot server VM version 1.8 update 25.

Synchrobench initializes the data structure by filling it up to a predefined *size* with values chosen randomly from a *range* $= \{1, 2, \ldots, 2 \times size\}$. It spawns from 1 to 72 threads. Running for 10 seconds, each thread repeatedly chooses one of the three operations with a fixed probability distribution function defined by the *update ratio* and executes it with an argument picked uniformly at random from the *range*.

The rationale behind the workload choice is to keep the size constant in expectation during the benchmark execution. This is the case because both the insert and remove values are chosen from the *range* that is twice the initial size, which means both will have a 50% chance to choose a value already in the list and same chance for the value to be absent. Note that the expected number of *effective* insert (inserting a value absent in the list) and *effective* remove (removing a value present in the list) is the half the update ratio.

Our experiments are done for list sizes in $\{100, 1000, 10000\}$, update ratios in $\{0\%, 10\%, 100\%\}$, and number of concurrent threads in $\{1, 2, 4, 8, 16, 24, \ldots, 72\}$. The results presented here are the average of 10 runs of 10 seconds for each point in the parameter space.

5.2 Results and evaluation

Figures 5 and 6 depict the number of operations per millisecond obtained on x86 and on SPARC, respectively. We only report the results for a list size of 100 on the SPARC architecture, since the curves on higher list sizes were similar for both architectures.

The left column of both figures is the contains-only workload, the right column is the update-only (insert and remove only) workload, and the middle column is a more realistic workload with 90% contains and 10% updates. Note that the level of contention increases from bottom to top as the list size decreases (leading to a higher chance of concurrent threads accessing the same node) and left to right as the number of write operations

increases. Also, since each operation has 50% chance to return false, the *effective* updates are roughly half of the shown percentage.

We also studied the contribution of *traversals* and *updates* in the execution time used by insert and remove in each algorithm. The *traversal* time is defined as the time between operation invocation and when it finds *prev* and *curr* where $prev.val < v \leq curr.val$ (including re-traversal time caused by abort and restart); for our algorithm, we also include the time of validate function. The *update* time is measured from just before locking to just after lock release; for Harris-Michael algorithm that does not use locks we simply measured the time taken by the *CAS* at the end of each update operation (excluding *CAS*es that happen during list traversal). Table 1 shows the relative execution time per operation per thread normalized by the lowest number among the data (shaded cell).

We can see that our new list algorithm outperforms both Harris-Michael's and the Lazy Linked List algorithms and remains scalable even under extremely heavy contention (the top-right corner of the throughput graphs). The only place where our algorithm drops in performance is when there are more threads than cores (above 64 threads, "core-saturation") and contention is high. This is an inherent problem to all lock-based algorithms: a thread holding a lock gets preempted from the CPU, while any other thread contending on the same lock cannot make progress even if it is assigned the CPU time. (We can see this from Table 1 that at 72 threads our update in the critical section took more than 100% longer than 64 threads.)

Comparison against Harris-Michael. Harris-Michael's algorithm in general scales well and performs really well under high contention and core saturation (at 72 threads). This can be explained by the fact that the algorithm is nonblocking: a thread preempted from the CPU at any time does not indefinitely hamper the progress of other threads.

We can see, however, on the left-hand side of the Figures 5 and 6 that even though the three algorithms feature the wait-free contains algorithm, our implementation of the Harris-Michael's contains is slower than the other two. The reason is the extra indirection needed when reading the *next* pointer in the combined *pointer-plus-boolean* structure. Note that the original C-like pseudocode of Harris [8] suggested the architecture-specific use of a bitmask on x86. While we could have done so in Java using sun.misc.Unsafe this is not recommended at it may annihilate the portability of the implementation. To avoid the overhead of reading an extra field when fetching the Java AtomicMarkableReference we implemented the run-time type identification (RTTI) variant with two subclasses that inherit from a parent node class and that represent the marked and unmarked states of the node as previously suggested [9]. This optimization requires, on the one hand, that a remove casts the subclass instance to the parent class to create a corresponding node in the marked state. It allows, on the other hand, the traversal to simply check the mark of each node by simply invoking instanceof on it to check the subclass the node instantiates.

From Table 1 we can see that Harris-Michael's algorithm has the most efficient updates because it only uses *CAS*, however it spends much longer on list traversal. We also found that above 40 threads, there is around 5% of the traversal time under 100% update workloads that is spent on attempts to unlink marked nodes during the traversal.

Comparison against the Lazy Linked List. The Lazy Linked List has almost the same performance as our algorithm under low contention (the left column and the bottom row in the graphs) because both share the same wait-free list traversal with zero overhead (as the sequential code does) and for the updates, when there is no interference from concurrent operations, the difference between our pre-locking-validation and Heller's post-locking-validation becomes negligible.

The difference raises however as the contention appears. The performance of the Lazy Linked List drops significantly due to its intense lock competition (as briefly explained in Section 1). By contrast, there are several features in our implementation that reduce the amount of contention on the locks significantly. For example, the pre-locking-validation that uses the versioned try-lock avoids aborting once the lock is acquired: if we cannot acquire the try-lock immediately because either the version has changed (meaning some concurrent thread has just modified the node) or the node is already locked (which means the version is going to change when it is unlocked), then we can already restart and try to read the new version. Another feature is that our insert and remove operations check the node value before locking so in case the update fails (because the value is present or absent), it returns with no particular overhead compared to contains. Table 1 shows the tremendous increase in execution time for the Lazy Linked List because of the contention on locks.

6 Related work

List-based sets. Heller *et al.* [9] proposed the Lazy Linked List algorithm, with a variety of optimizations. In particular, they mentioned doing a validation prior to locking, and using a single lock within an insert operation. One of the reasons why our implementation is faster than the Lazy Linked List is the use of a new *versioned try-lock* mechanism (hinted in [16] for the TM context) that allows validating before acquiring the lock.

Harris [8] proposed a non-blocking linked list algorithm that splits the removal of a node into two atomic steps: a logical deletion that marks the node and a physical removal that unlinks the node from the structure. Michael [15] proposed advanced memory reclamation algorithms for Harris' algorithm. In our implementation, we rely on Java's garbage collector for memory reclamation [17].

For a comprehensive survey of list-based sets, we refer to the textbook of Herlihy and Shavit [12].

Concurrency metrics. Sets of accepted schedules are commonly used as a metric of concurrency provided by a shared-memory implementation. For static database transactions, Kung and Papadimitriou [14] use the metric to capture the parallelism of a locking scheme. While acknowledging that the metric is theoretical, they insist that it may have "practical significance as well, if the schedulers in question have relatively small scheduling times as compared with waiting and execution times."

Herlihy [10] employed the metric from [14] to compare various optimistic and pessimistic synchronization techniques using commutativity of operations constituting high-level transactions. A synchronization technique is implicitly considered in [10] as highly concurrent, namely "optimal", if no other technique accepts more schedules. By contrast, we focus here on a *dynamic* model where the scheduler cannot use the prior knowledge of all the shared addresses to be accessed. Optimal concurrency can thus be seen as a variant of *permissiveness*, originally defined for opaque TM [6], applied to the case of dynamic data structures with high-level sequential semantics.

In the TM context, Gramoli et al. [5] defined a concurrency metric, the *input acceptance*, as the ratio of committed transactions over aborted transactions for a given schedule. Unlike our metric, input acceptance does not apply to lock-based programs.

7 Conclusion

Intuitively, the ability of an implementation to successfully process interleaving steps of concurrent threads is an appealing property that should be met by performance gains. In this paper, we support this intuition by presenting a concurrency-optimal list-based set that outperforms (less concurrent) state-of-the-art algorithms. Does the claim also hold for other data structures? We suspect so. For example, similar but more general data structures, such as skip-lists or tree-based dictionaries, may allow for optimizations similar to the ones proposed in this paper.

References

[1] H. Attiya, E. Hillel, and A. Milani. Inherent limitations on disjoint-access parallel implementations of transactional memory. In *SPAA*, pages 69–78, 2009.

[2] H. Attiya and J. Welch. *Distributed Computing. Fundamentals, Simulations, and Advanced Topics.* John Wiley & Sons, 2004.

[3] F. Ellen, P. Fatourou, E. Kosmas, A. Milani, and C. Travers. Universal constructions that ensure disjoint-access parallelism and wait-freedom. In *PODC*, pages 115–124, 2012.

[4] V. Gramoli. More than you ever wanted to know about synchronization. In *PPoPP*, Feb 2015.

[5] V. Gramoli, D. Harmanci, and P. Felber. On the input acceptance of transactional memory. *Parallel Processing Letters*, 20(1):31–50, 2010.

[6] R. Guerraoui, T. A. Henzinger, and V. Singh. Permissiveness in transactional memories. In *DISC*, pages 305–319, 2008.

[7] R. Guerraoui and M. Kapalka. *Principles of Transactional Memory*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2010.

[8] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *DISC*, pages 300–314, 2001.

[9] S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. N. Scherer, and N. Shavit. A lazy concurrent list-based set algorithm. In *OPODIS*, pages 3–16, 2006.

[10] M. Herlihy. Apologizing versus asking permission: optimistic concurrency control for abstract data types. *ACM Trans. Database Syst.*, 15(1):96–124, 1990.

[11] M. Herlihy. Wait-free synchronization. *ACM Trans. Prog. Lang. Syst.*, 13(1):123–149, 1991.

[12] M. Herlihy and N. Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2008.

[13] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[14] H. T. Kung and C. H. Papadimitriou. An optimality theory of concurrency control for databases. In *SIGMOD*, pages 116–126, 1979.

[15] M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *SPAA*, pages 73–82, 2002.

[16] T. Riegel, C. Fetzer, and P. Felber. Time-based transactional memory with scalable time bases. In *SPAA '07*, pages 221–228, New York, NY, USA, 2007. ACM.

[17] Sun Microsystems. *Memory Management in the Java HotSpot Virtual Machine*, April 2006. `http://www.oracle.com/technetwork/java/javase/memorymanagement-whitepaper-150215.pdf`.

[18] H. Sutter. Choose concurrency-friendly data structures. *Dr. Dobb's Journal*, June 2008.

A Sequential implementation of the set type

```
1:  Shared variables:
2:    Initially head, tail,
3:      head.val = −∞, tail.val = +∞
4:      head.next = tail
5:  insert(v):
6:    prev ← head                                  ▷ copy the address
7:    curr ← read(prev.next)                       ▷ fetch the next element
8:    while (tval ← read(curr.val)) < v do
9:      prev ← curr
10:     curr ← read(curr.next)                     ▷ fetch from memory
11:   if tval ≠ v then                             ▷ tval is stored locally
12:     X ← new-node(v, prev.next)                 ▷ v and address of curr
13:     write(prev.next, X)                        ▷ next points to the new element
14:   return (tval ≠ v)
15: remove(v):
16:   prev ← head                                  ▷ copy the address
17:   curr ← read(prev.next)                       ▷ fetch next field
18:   while (tval ← read(curr.val)) < v do         ▷ val local copy
19:     prev ← curr
20:     curr ← read(curr.next)
21:   if tval = v then
22:     tnext ← read(curr.next)                    ▷ fetch the node after curr
23:     write(prev.next, tnext)                    ▷ delete the node
24:   return (tval = v)
25: contains(v):
26:   curr ← head
27:   curr ← read(prev.next)
28:   while (tval ← read(curr.val)) < v do
29:     curr ← read(curr.next)
30:   return (tval = v)
```

Algorithm 3: Sequential implementation *LL* (*sorted linked list*) of *set* type

An object of the *set* type stores a set of integer values, initially empty, and exports operations insert(v), remove(v), contains(v); $v \in \mathbb{Z}$. The update operations, insert(v) and remove(v), return a boolean response, true if and only if v is absent (for insert(v)) or present (for remove(v)) in the list. After insert(v) is complete, v is present in the list, and after remove(v) is complete, v is absent in the list. The contains(v)} returns a boolean a boolean, true if and only if v is present in the list.

The sequential implementation *LL* of the *set* type is presented in Algorithm 3. The implementation uses a *sorted linked list* data structure in which each element (except the *tail*) maintains a *next* field to provide a

pointer to the successor node. Initially, the *next* field of the *head* element points to *tail*; *head* (resp. *tail*) is initialized with values $-\infty$ (resp. $+\infty$) that is smaller (resp. greater) than the value of any other element in the list.

B Proof of Theorem 1

Theorem 1 (Correctness)**.** *Versioned List is linearizable with respect to the set type.*

Proof. We show that $\tilde{S}$ defined in Section 3 is consistent with the sequential specification of type *set*. When we refer to $\mathsf{read}(X)$, where X is a node, we mean the first read of a node's field.

Let $\tilde{S}^k$ be the prefix of $\tilde{S}$ consisting of the first k complete operations. We associate each $\tilde{S}^k$ with a set q^k of objects that were successfully inserted and not subsequently successfully removed in $\tilde{S}^k$. We show by induction on k that the sequence of state transitions in $\tilde{S}^k$ is consistent with operations' responses in $\tilde{S}^k$ with respect to the *set* type.

The base case $k = 1$ is trivial: the *tail* node containing $+\infty$ is successfully inserted. Suppose that $\tilde{S}^k$ is consistent with the *set* type and let π_1 with argument $v \in \mathbb{Z}$ and response r_{π_1} be the last operation of $\tilde{S}^{k+1}$. We want to show that $(q^k, \pi_1, q^{k+1}, r_{\pi_1})$ is consistent with the *set* type.

(1) Let $\pi_1 = \mathsf{insert}(v)$ return true in $\tilde{S}^{k+1}$. We show below that each preceding $\pi_2 = \mathsf{insert}(v)$ returning true is followed by $\mathsf{remove}(v)$ returning true, such that $\pi_2 \rightarrow_{\tilde{S}^{k+1}} \mathsf{remove}(v) \rightarrow_{\tilde{S}^{k+1}} \pi_1$. Suppose the opposite. Observe that π_1 performs its penultimate read on a node X that stores a value $v' < v$ and the last read is performed on a node that stores a value $v'' > v$. By construction of $\tilde{S}$, π_1 is linearized at the write on node X in Line 45. Observe that π_2 must also perform a write to the node X (otherwise it is easy to see that one of π_1 or π_2 would return false). By assumption, the write to X in shared-memory by π_2 in Line 45 precedes the corresponding write to X in shared-memory by π_1. But π_1 can return true from the *cas* performed in Line 43 only after π_2 releases the versioned lock on X_1 by performing the event in Line 46. Thus, π_1 could not have returned true—a contradiction.

Let $\pi_1 = \mathsf{insert}(v)$ return false in $\tilde{S}^{k+1}$. We show that there exists a preceding $\pi_2 = \mathsf{insert}(v)$ returning true that is not followed by $\pi_3 = \mathsf{remove}(v)$ returning true, such that $\pi_2 \rightarrow_{\tilde{S}^{k+1}} \pi_3 \rightarrow_{\tilde{S}^{k+1}} \pi_1$. Suppose that such a π_2 does not exist. Thus, π_1 must perform its last read on a node X that stores value $v'' > v$, acquire the versioned lock on X (Line 43) and return true—a contradiction to the assumption that π_1 returned false.

It is easy to verify that the conjunction of the above two claims proves that $\forall q \in Q$; $\forall v \in \mathbb{Z}$, $\tilde{S}^{k+1}$ satisfies $(q, \mathsf{insert}(v), q \cup \{v\}, (v \notin q))$.

(2) If $\pi_1 = \mathsf{remove}(v)$, similar arguments as applied to $\mathsf{insert}(v)$ prove that $\forall q \in Q$; $\forall v \in \mathbb{Z}$, $\tilde{S}^{k+1}$ satisfies $(q, \mathsf{remove}(v), q \setminus \{v\}, (v \in q))$.

(3) Let $\pi_1 = \mathsf{contains}(v)$ return true in $\tilde{S}^{k+1}$. We show that there exists $\pi_2 = \mathsf{insert}(v)$ returning *true* that is not followed by any $\mathsf{remove}(v)$ returning *true*, such that $\pi_2 \rightarrow_{\tilde{S}^{k+1}} \mathsf{remove}(v) \rightarrow_{\tilde{S}^{k+1}} \pi_1$. Recall that π_1 is linearized at the last read of an node, say X, performed by π when π reads the *deleted* field of X to be false (Line 18). By the algorithm, there exists $\pi_2 = \mathsf{insert}(v)$ such that $\pi_2 \rightarrow_{\tilde{S}^{k+1}} \pi_1$ (let π_2 be the latest such operation). Suppose that there exists a $\mathsf{remove}(v)$ that returns *true*, such that $\pi_2 \rightarrow_{\tilde{S}^{k+1}} \mathsf{remove}(v) \rightarrow_{\tilde{S}^{k+1}} \pi_1$. Thus, $\mathsf{remove}(v)$ performs the write event in Line 56 prior to the read of $X.deleted$ by π_1. But then π_1 must read $X.deleted$ to be true and return false—a contradiction.

Now, let $\pi_1 = \mathsf{contains}(v)$ return false in $\tilde{S}^{k+1}$. Thus, (1) there exists a $\pi_2 = \mathsf{remove}(v)$ returning *true* that is not followed by any $\mathsf{insert}(v)$ returning *true*, such that $\pi_2 \rightarrow_{\tilde{S}^{k+1}} \mathsf{insert}(v) \rightarrow_{\tilde{S}^{k+1}} \pi_1$, or (2) there does not exist any $\mathsf{insert}(v)$ returning true such that $\mathsf{insert}(v) \rightarrow_{\tilde{S}^{k+1}} \pi_1$. We consider two cases:

- Suppose that π_1 reads $(X.value \neq v)$ in Line 18, where X is the last node read by π_1 in α. Thus, ℓ_{π_1} is assigned to the read of the *next* field of the node, say X' accessed by π_1 immediately before X. Assume by contradiction that there exists $\pi_2 = \mathsf{insert}(v)$ that returns *true* such that there does not exist any $\mathsf{remove}(v)$ that returns *true*; $\pi_2 \rightarrow_{\tilde{S}^{k+1}} \mathsf{remove}(v) \rightarrow_{\tilde{S}^{k+1}} \mathsf{contains}(v)$. But then π_1 must read $(X.value = v)$ in Line 18 and return true—contradiction.

- Suppose that π_1 reads $(X.value = v)$ and $X.deleted$ to be true in Line 18. Clearly, there exists a $\pi_2 = \mathsf{remove}(v)$ that is concurrent to π_1 and returns true in $\tilde{H}$. By the assignment of linearization points, ℓ_{π_1} is assigned to the first event performed by π_2 immediately after the write to $X.deleted$, but prior to the read

of $X.deleted$ by π_1, where X is the last node read by π_1. We consider two cases: (1) Suppose that some such event of π_2 exists. We claim that there does not exist any $\pi_3 = \mathsf{insert}(v)$ that returns true such that $\pi_2 \rightarrow_{\tilde{S}^{k+1}} \pi_3 \rightarrow_{\tilde{S}^{k+1}} \pi_1$. Any such π_2 must acquire the versioned lock on X' (Line 43), the node read by π_1 immediately prior to X. Since π_1 reads $(X.value = v)$ and $X.deleted$ to be true, π_2 must also acquire the versioned lock on X' (Line 53). By our assumption, $\ell_{\pi_2} \rightarrow_{\tilde{S}^{k+1}} \ell_{\pi_3}$. Thus, π_3 acquires the versioned lock on X' only after π_2 releases it in Line 59. But we linearize π_1 prior to ℓ_{π_3} by choosing it to be the event performed by π_2 in Line 57—a contradiction to our assumption that $\ell_{\pi_3} <_\alpha \ell_{\pi_1}$. (2) Otherwise, if no such event of π_2 exists, ℓ_{π_1} is chosen as the read of $X.deleted$ by π_1. Since π_2 does not release the versioned lock on X' prior to the read of $X.deleted$ by π_1, there does not exist any $\mathsf{insert}(v)$ that returns true such that $\mathsf{insert}(v) \rightarrow_{\tilde{S}^{k+1}} \pi_1$. Now, by the assignment of linearization points, $\pi_2 \rightarrow_{\tilde{S}^{k+1}} \pi_1$.

Thus, inductively, the sequence of state transitions in $\tilde{S}$ satisfies the sequential specification of the *set* type. □